On Rearrangement of Items Stored in Stacks

Mario Szegedy and Jingjin Yu

Department of Computer Science, Rutgers University
{szegedy, jingjin.yu}@cs.rutgers.edu

Abstract. There are $n \geq 2$ stacks, each filled with d items, and one empty stack. Every stack has capacity $d > 0$. A robot arm, in one stack operation (step), may pop one item from the top of a non-empty stack and subsequently push it onto a stack not at capacity. In a *labeled* problem, all nd items are distinguishable and are initially randomly scattered in the n stacks. The items must be rearranged using pop-and-pushs so that in the end, the k^{th} stack holds items $(k-1)d+1, \ldots, kd$, in that order, from the top to the bottom for all $1 \leq k \leq n$. In an *unlabeled* problem, the nd items are of n types of d each. The goal is to rearrange items so that items of type k are located in the k^{th} stack for all $1 \leq k \leq n$. In carrying out the rearrangement, a natural question is to find the least number of required pop-and-pushes.

Our main contributions are: (1) an algorithm for restoring the order of n^2 items stored in an $n \times n$ table using only $2n$ column and row permutations, and its generalization, and (2) an algorithm with a guaranteed upper bound of $O(nd)$ steps for solving both versions of the stack rearrangement problem when $d \leq \lceil cn \rceil$ for arbitrary fixed positive number c. In terms of the required number of steps, the labeled and unlabeled version have lower bounds $\Omega(nd + nd\frac{\log d}{\log n})$ and $\Omega(nd)$, respectively.

1 Introduction

In a range of real-world applications, items are arranged in *stacks* to balance between efficient space usage and ease of storage and retrieval. In a stack based storage solution, only the item on the top of an non-empty stack can be accessed instantaneously. If other stored items are to be retrieved, additional items must be moved beforehand. Such an approach, while preventing the direct random access of an arbitrary item, allows more economical utilization of the associated storage space, which is always limited. A prime example is the stacking of containers at shipping ports [3, 7], where stacks of container may need to be rearranged (shuffled) for retrieval in a specific order. Similar scenarios also appear frequently elsewhere, e.g., parking yards during busy hours in New York City, the re-ordering of misplaced grocery items on supermarkets shelves [15], the rearrangement of goods in warehouses [5], and so on. In all these application

scenarios, the overall efficiency of the system critically depends on minimizing the number of item storage and retrieval operations.

We are thus motivated to examine the *stack rearrangement* problem in which there are n stacks (i.e., LIFO queues), each filled to capacity with d items. In the *labeled* version, or `LSR` (labeled stack rearrangement), the items in the stacks are uniquely labeled $1, \ldots, nd$. Given an arbitrary initial arrangement of the items, we would like to rearrange them to follow lexicographic order, in which the k^{th} stack, $1 \leq k \leq n$, contains items labeled $(k-1)d+1$ to kd, with numbers increasing monotonically from the top of the stack to the bottom of the stack. In a single *pop-and-push*, *step*, or *stack operation* (we use these terms interchangeably in this paper), an item can be popped off from any non-empty stack and immediately pushed onto a stack which is not filled to its capacity d. To allow the rearrangement of items, we assume that there is an empty buffer stack with capacity d. During the moves the buffer can hold items but it must be emptied by the end. Our goal is to *minimize* the number of pop-and-pushes to take the stacks from an arbitrary initial arrangement to the specified target arrangement, which is equivalent to having an arbitrary goal arrangement.

In an *unlabeled* version, or `USR` (unlabeled stack rearrangement), we still require that items labeled $(k-1)d+1, \ldots, kd$ go into k^{th} stack but do not require these items take a specific order within the stack. This is equivalent to saying that we would like to sort nd items with n types of d each so that the k^{th} stack contains only items of type k. (see Fig. 1).

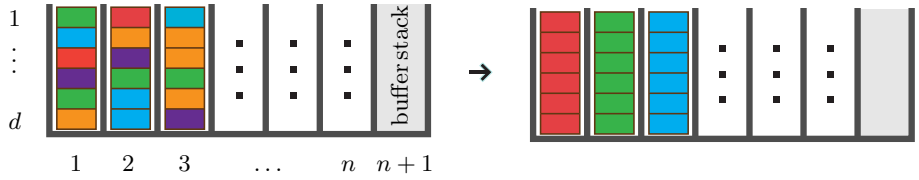

Fig. 1. An illustration of the `USR` problem with an initially empty buffer stack. [left] An initial arrangement of the items. [right] A sorted target arrangement. In `LSR`, items within the k^{th} stack are further labeled $(k-1)d+1, \ldots, kd$ with the smaller labeled items closer to the top of the stack in the goal/target arrangement.

The stack rearrangement problem was first formally studied in the stated form in [15], in which an $O(nd \max\{\log n, \log d\})$ upper bound is established. Heuristics-based search methods are also developed that can compute the optimal solution for stack rearrangement problems involving tens of items. A closely related problem is the Hanoi tower problem [4, 13, 25], which has additional constraints limiting the relative order of items in a stack during the rearrangement process.

In the robotics domain, our study relates to multi-object rearrangement tasks, which may be carried out using mobile robots [2, 11, 17] or fixed robot arms [16, 18–20]. Clearly a challenging task and motion planning problem in the general

setting [18], even the combinatorial aspect of object rearrangement is shown to be computationally hard in multiple problems in seemingly simple setups [16]. A multi-arm rearrangement problem is recently explored [21]. In a more abstract setting, multi-object rearrangement has also been studied under the PushPush line of problems [8, 9]. More broadly, object rearrangement problems are connected to multi-robot motion planning problems [10, 22, 23, 29] and the problem of navigation among movable obstacles [24, 26, 28]. Lastly, as a sorting problem, our study share some similarities with sorting networks [1, 27].

Our main algorithmic results on the stack rearrangement problem are:

- For an average case, $\Omega(nd + nd\frac{\log d}{\log n})$ steps are necessary for `LSR` (Lemma 1) and $\Omega(nd)$ steps are necessary for `USR` (Lemma 2).
- For any fixed integer $m > 2$, `LSR` (and therefore, `USR`) with $d \leq n^{\frac{m}{2}}$ can be solved using $O(nd)$ steps. If m is an input parameter instead, `LSR` with $d = n^{\frac{m}{2}}$ can be solved using $O(3^m nd)$ steps (Theorem 1). Therefore, for an arbitrary fixed real number c, `LSR` may be solved using $O(nd)$ steps for $d \leq \lceil cn \rceil$ (Theorem 2).

As an intermediate step toward solving `USR` and `LSR`, we investigated a permutation problem which we call the *Rubik table problem*. The task in a Rubik table problem is to reach an arbitrary permutation on an $n \times n$ table with n^2 unique items using a small number of column/row permutations where each permutation may arbitrarily rearrange items within a single table column or row. Further generalizations allow additional dimensions to be added to the table. The results (Propositions 1–3 in Section 3) are of independent interest for global coordination problems involving data and physical objects.[1]

The rest of the paper is organized as follows. In Section 2, we provide a a lower bound for `USR` and `LSR` for completeness. In Section 3, we define and examine Rubik table problems. In Section 4, upper bounds are established for `USR` and `LSR`. We discuss and conclude in Section 5.

2 Lower Bounds for Stack Rearrangement

It takes at least $\Omega(nd)$ steps to solve the stack rearrangement problem for a typical input instance, because most items must move at least once to get into place. In this section, we prove a stronger lower bound. We mention that similar bounds are described in [15]. We provide a more accurate bound for `LSR` here with a proof counting the number of bits required to describe an algorithm. A bound for `USR` is also included for completeness.

[1] A version of the result enabled a result on high-dimensional multi-robot motion planning on grids [29]. We note that [29] explicitly cited this (then unpublished) work and the intersection between [29] and this work is insignificant.

Lemma 1 (Lower Bound for LSR). *Any algorithm for LSR must take at least* $\Omega(nd + nd\frac{\log d}{\log n})$ *steps for an average input.*

Proof. The proof is by a counting argument. Any correct algorithm must follow different paths for all of the $(nd)!$ initial arrangements, since two different initial arrangements followed by identical moves would lead to different final arrangements. A step of the algorithm can be described with $2\lceil\log(n+1)\rceil$ bits: (from where, to where). Therefore, the two-based logarithm of the number of possible sequences of at most t steps is upper bounded by $2t\lceil\log(n+1)\rceil$. So as long as it holds that

$$2t\lceil\log(n+1)\rceil \leq \log\left(0.01 \cdot (nd)!\right) = \Omega(nd\log nd),$$

i.e. when $t = o(nd + nd\frac{\log d}{logn})$, the initial arrangements that can be solved with t steps constitute only a small minority of all arrangements. The counter-positive of this gives the lemma. □

Lemma 2 (Lower Bound for USR). *Any algorithm for USR must take at least* $\Omega(nd)$ *steps for an average input.*

Proof. Me may view the generation of a random instance as selecting from n types of items with replacement d for up to nd rounds. Therefore, there are $(\Theta(n))^{\Theta(nd)}$ initial configurations. Following the same argument from the proof of Lemma 1, $\Omega(nd)$ steps are necessary. □

3 The Rubik Table Problem

In tackling the stack rearrangement problems, we encountered a table shuffling problem of independent interest. We call it the *Rubik table problem*, which allows globally coordinated token swapping operations to be efficiently carried out. We associate it with the name *Rubik* as it shares some similarity with the Rubik's Cube toy (Fig. 2). The basic setting deals with a planar table.

Problem 1 (The Rubik Table Problem)**.** *Let M be an $n \times n$ table containing n^2 unique items, one in each cell of the table. In a shuffle operation, the items in a single row or a single column of M may be permuted in an arbitrary manner. Given two configurations X_I and $X_G = \pi(X_I)$ of the items where π is some arbitrary permutation over n^2, provide a sequence of shuffles that takes the table M from X_I to X_G.*

It appears that at least $2n$ shuffles are required for solving a Rubik table problem in an average case, since, conservatively, each row and column needs to be permuted at least once with high probability. We show that an upper bound of $3n$ shuffles is possible, closely matching the lower bound.

Proposition 1 (Linear Shuffle Algorithm for Rubik Table Problem). *An arbitrary Rubik table problem is solvable using* $3n$ *row/column shuffles.*

Before presenting the proof of Proposition 1, we introduce a Kőnig-Hall type matching theorem [14] with parallel edges.

Lemma 3 (Hall's Matching Theorem with Parallel Edges). *Let* B *be a* d*-regular* ($d > 0$) *bipartite graph on* $n + n$ *nodes, possibly with parallel edges. Then* B *has a perfect matching.*

Proof. Let the vertex set of B be $L \dot{\cup} R$, where L is the left partite site of B and R is the right partite set. Consider a maximal matching M in B. We show that M meets all of the vertices of B, so it is perfect. Assume that M is not incident to some vertex $v \in L$. Consider all nodes of B reachable by an alternating path from v, that is, a path that starts in v, goes to R along some edge, then goes back to L along an edge of M (if such an edge exists), then goes along an arbitrary edge to R, an so on, always alternating between edges of M and non-edges of M. We stop whenever we want. If any such path P ends up in a point w of R not matched in M, we could make M bigger as follows: We discard from M its intersection with P and add $P \backslash M$ to it, creating

$$M' = M \triangle P = (M \backslash (M \cap P)) \cup (P \backslash M)$$

It is easy to see that M' is a matching and $|M'| = |M| + 1$, contradicting the maximality of M. Otherwise, let $L' \subseteq L$ be the subset of L reachable via an alternating path from v (which includes v too), and let $R' \subseteq R$ be the set of nodes reachable with alternating path from v. Then $|L'| > |R'|$, since every vertex in R' has a matching partner in L' through M, and in addition L' contains v, which is not a partner of any node in R'. Furthermore, all neighbors of the nodes in L' must be in R', otherwise we could find a neighbor w of a node t in L', which is reachable via an alternating path from v (formed by adding edge (t, w) to the alternating path from v to t), but unmatched in M. Since the nodes in L' have a total of $d|L'|$ edges incident to them (counted with multiplicities), which is more than the number $d|R'|$ of edges incident to R' (counted with multiplicities), we again have a contradiction. □

Proof of Proposition 1. The $n + n + n$ shuffles to construct an arbitrary permutation π are outlined in Table 1.

The preparation phase is necessary for the column fitting. We need to prove that we can permute the items only within every column (i.e. such that no item changes column coordinate) with the effect that the n items destined to go to any fixed column end up in n different rows. This comes from Lemma 4, which shows the feasibility of the preparation phase and therefore, the entire algorithm. □

1. Preparation:	By appropriately permuting the items within each column we reach the situation where the n items destined to go to any fixed column will end up in n different rows.
2. Column fitting:	By appropriately permuting the items within each row we reach the situation where the n items destined to go to any fixed column goes to that column.
3. Row Fitting:	By appropriately permuting the items within each column we move each item into its final destination.

Table 1. A three-phase shuffle plan for rearranging items in an $n \times n$ Rubik table.

Lemma 4. *Let M be an $n \times n$ matrix filled with items of n different types. The number of items of types i is exactly n for $1 \leq i \leq n$. Then we can permute the items within each column of M separately such that in the resulting new arrangement all of the n items of any fixed type i (for $1 \leq i \leq n$) go into separate rows. In other words, the resulting arrangement is a Latin square.*

Proof. We begin by creating a bipartite graph $B(T, C)$ on $n+n$ nodes such that the left partite set, T, stands for all the types $\{1, \ldots n\}$, and the right partite set, C, stands for all the columns of M. We draw k edges between type j and column i, if column i contains k items of type j. Notice that B is n-regular from both sides with parallel edges. Lemma 3 implies that graph B contains a perfect matching M_1. Label the edges of this matching with the number 1, and take it out of B. We obtain an $(n-1)$-regular bipartite graph on which Lemma 3 may be applied again. We keep creating matchings M_2, M_3, ..., in this fashion and label their edges with 2, 3, ..., until we arrive at M_n, when we stop. Notice that now each type $j \in T$ is connected to edges labeled with 1 through n, and that each column C_i is connected to all n types of edges as well (in both cases exactly one from each type). For every $1 \leq i \leq n$ we rearrange the items in column C_i such that the item corresponding to an edge labeled with i goes into the i^{th} row. There will be no collisions by construction and we have arrived at the desired arrangement. □

From an algorithmic perspective, each matching step in Lemma 4 can be done in expected $n \log n$ time [12]. Alternatively, if a deterministic algorithm is desirable, a matching can be computed in $O(n^2)$ time [6]. The n matchings can then be completed in $O(n^2 \log n)$ expected time or $O(n^3)$ deterministic time.

To provide intuition, Fig. 2 illustrates an application of Proposition 1 on a 4×4 table containing 4 types of items. After n column shuffles and n row shuffles, Fig. 2(a) $\rightarrow$ Fig. 2(d) is achieved. It is clear that with one more round of column shuffles after Fig. 2(d), items within each type, if distinguishable, can be sorted into arbitrary order. As a general global coordination scheme, Proposition 1 turns out to be applicable to multi-robot motion planning tasks [29].

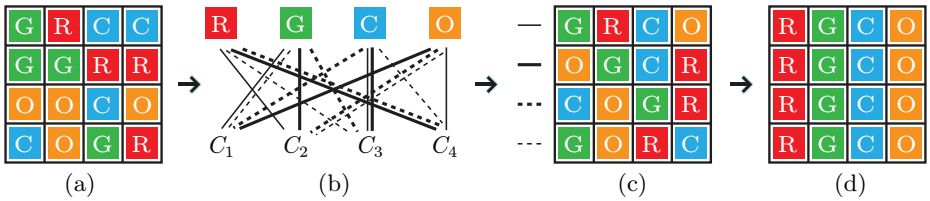

Fig. 2. Illustration of applying the first two phases in Proposition 1. (a) Initial 4×4 table and a random arrangement of 4 types of colored (red, green, cyan, orange) items. (b) The bipartite graph constructed from the table and a possible set of 4 perfect matchings, where $C_i, 1 \leq i \leq 4$ are the columns. As an example, green appears twice in the the first column of the table in (a) so there are two edges between green and C_1. Each matching is marked with a unique line type (thin, thick, thick dash, thin dash). (c) Permuting each column according to the matching results in each row containing each item type exactly once. (d) Permuting each row of (c) then sorts all columns.

It can be readily verified that Proposition 1 can be generalized to tables that are not squares.

Corollary 1 (Linear Shuffle Algorithm for Rubik Rectangle Problem). *Let M be an $n \times m$ table filled with nm unique items. In n row shuffles and $2m$ column shuffles, items in M can be sorted arbitrarily.*

For stack rearrangement problems, a more involved version of Proposition 1 is required to support table cells with depth. For that, we observe the algorithm for the Rubik table problem holds when the table has an additional dimension. That is, we may allow M to have a "depth" K and in each row or column permutation, nK items are arranged arbitrarily. This version of the Rubik table problem is denoted as the *fat Rubik table problem*.

Problem 2 (Fat Rubik Table Problem)**.** *Let M be an $n \times n \times K$ (row $\times$ column $\times$ depth) table containing n^2K unique items, one in each cell of the table. In a shuffle operation, the items in a single fat row (i.e., items with indices in $\{i\} \times \{1, \ldots, n\} \times \{1, \ldots, K\}$ for $1 \leq i \leq n$) or a single fat column (i.e., items with indices in $\{1, \ldots, n\} \times \{i\} \times \{1, \ldots, K\}$ for $1 \leq i \leq n$) of M may be permuted in an arbitrary manner. Given two configurations X_I and $X_G = \pi(X_I)$ of the items where π is some arbitrary permutation over $1, \ldots, n^2K$, provide a sequence of shuffles that takes the table M from X_I to X_G.*

Proposition 2 (Linear Shuffle Algorithm for Fat Rubik Table Problem). *The fat Rubik table problem may be solved in $3n$ shuffles.*

Proof. The proof of Proposition 1 can be adapted with minor changes. A similar three-phase procedure will be followed. Again, the crucial part is the proof of the preparation phase, where we show that we can permute the items within each

fat column to reach the situation where the nK items destined to go to any fixed fat column will end up in nK positions, that are different when we project them to the first and third coordinates. The needed procedure for doing this provided in Lemma 5. □

Lemma 5. *Let M be an $n \times n \times K$ table (row × column × depth) filled with items of n different types. The number of items of types j is exactly nK for $1 \leq j \leq n$. Then we can permute the items within each fat column $(*, i, *)$ of M $(1 \leq i \leq n)$ such that for any fixed type j $(1 \leq j \leq n)$, if we look at the nK items of type j, they occupy distinct (row, depth) values when we project the triplet representing their new positions to the pair of row and depth coordinates.*

Proof. The proof of the lemma is again based on applying Lemma 3 on an $n+n$ bipartite graph. The nodes on the left are n different types and the nodes on the right represent the fat columns. The edges correspond to the items, end we have K parallel edges between right node i and left node j as long as K items need to go from fat column i to fat column j. The only difference is that now the graph is nK-regular rather than n-regular. Again, we can decompose the edge-set of this bipartite graph into nK perfect matchings in an iterative manner, which gives the solution we are looking for. □

Fig. 3 illustrates an application of Proposition 2 to derive the the first two sets of permutations for restoring order to a $3 \times 3 \times 3$ fat Rubik table. In applying Lemma 5, type j corresponds to items numbered $(j-1)*3$ to $j*3-1$. For example, all items numbered $1-3$ are treated as type 1.

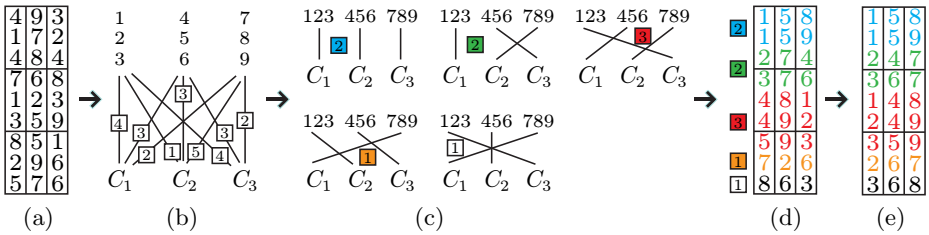

Fig. 3. Illustration of applying the first two shuffle phases from Proposition 2. (a) Initial $3 \times 3 \times 3$ fat Rubik table and a random arrangement of 9 types of items. (b) The (weighted) regular bipartite graph from the setup in (a). The numbers on edges denote the wegiht/multiplicity of the edges. (c) The 5 sets of (weighted) perfect matching extracted from (b). (d) The fat-column permutations based on the matchings. Note that the numbers in the columns remain the same between (a) and (d). (e) The following fat-row permutations which correctly sort the columns. With one more round of fat-column permutations, we can sort the table so that each cell contains a single item type. Additional distinguishability within a cell is allowed as well.

Again, non-square fat Rubik tables can be supported. We omit the details.

Lastly, we present a high-dimensional version of the Rubik table problem. A fat version is again possible, which we do not future detail here.

Proposition 3 (Rubik R-D Table Problem). *Let M be an $\underbrace{n \times \ldots \times n}_{R}$ table, $R \geq 2$, filled with n^R unique items. Assuming that any $(R-1)$-dimensional column can be arbitrarily shuffled, then M can be arbitrarily sorted in $(2^R - 1)n^{R-1}$ shuffles.*

Proof. Let $F(n, R)$ be the number of shuffles for given n and R. We prove claimed bound on $F(n, R)$ by induction on R. We can do the 2 dimensional case in $3n$ shuffles by Proposition 1. For $R > 2$, select out the first two dimensions and treat the remaining $R - 2$ dimensions as the depth of a fat Rubik table. By the induction hypothesis we can permute any fat column of M any way we want in $F(n, R-1)$ shuffles (by the induction hypothesis, $F(n, R-1) = (2^{R-1} - 1)n^{R-2}$). In the preparation phase we must do n of these. Then we do n^{R-1} row operations and finally we do again permutations on the fat columns, which costs $nF(n, R-1)$. Altogether, we have

$$F(n, R) = 2nF(n, R-1) + n^{R-1} = 2n(2^{R-1} - 1)n^{R-1} = (2^R - 1)n^{R-1}.$$

□

4 Tighter Upper Bounds for Stack Rearrangement

Results on fat Rubik table problem leads to significantly improved upper bounds for `USR` and `LSR` that largely match the lower bound (asymptotically), which we establish in this section. The proposed algorithmic approach applies directly to `LSR` and therefore `USR`. The improved upper bounds are obtained through recursive applications of the fat Rubik table result (Proposition 2) through "simulated" fat Rubik table column and row permutations. The recursion is done based on increasing $2\frac{\log d}{\log n}$. We first address the case of $2\frac{\log d}{\log n} \leq 1$ (i.e., $d \leq \sqrt{n}$), followed by the case $2\frac{\log d}{\log n} \leq 2$ (i.e., $d \leq n$), and finally the general case of $2\frac{\log d}{\log n} \leq m$ (i.e., $d \leq n^{\frac{m}{2}}$).

4.1 Linear Step Algorithm for `LSR` with $d \leq \sqrt{n}$

We first examine `LSR` where $d = \sqrt{n}$.

Lemma 6 (Linear Step Algorithm for `LSR`, $d = \sqrt{n}$). *`LSR` with $d = \sqrt{n}$ can be solved using $O(nd)$ steps.*

Proof. We construct an $n' \times n' \times K$ fat Rubik table with $n' = K = d = \sqrt{n}$. A depth $K = d$ fat cell of the table with index $(i, j), 1 \leq i, j \leq n' = d$ is identified with the stack indexed $(j-1) * d + i$ (see Fig. 4 for an example), which ranges between 1 and $n = d^2$.

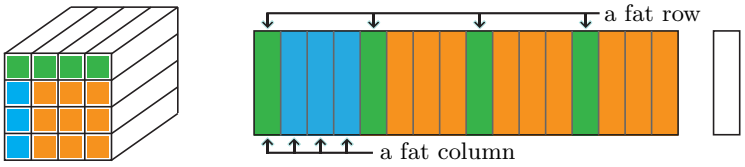

Fig. 4. Correspondence between a $d \times d \times d$ fat Rubik table and the $n = d^2$ stacks of depth d in a stack rearrangement problem instance. $d = 4$.

We first show that we can simulate a single fat column permutation of $n'K = \sqrt{n}d = d^2$ items in $O(d^2)$ stack operations, which can be achieved by:

1. Moving the content of $\sqrt{n} = d$ stacks to the top of the n stacks using $O(d^2)$ steps. For each stack, we may move its content to the top of other stacks using the operations illustrated in the first four figures in Fig. 5, which takes $3d$ steps (we left out some minor ordering details that can be easily filled in by the reader). Applying this to d stacks requires $3d^2$ steps, resulting the configuration shown in the fifth figure of Fig. 5.

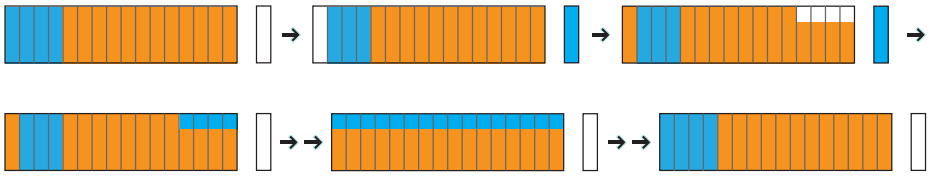

Fig. 5. Illustration of the steps for realizing a simulated fat column permutation in $O(n'K) = O(d^2)$ steps. The cyan stacks are the d stacks of interest. First step (indicated by the arrow) illustrates emptying the leftmost stack to the buffer. Then, the top of some stacks not of current interest (the orange ones) can be moved to the emptied stack (second step). Subsequently, the buffer content can be put on the top of stacks (third step). After this is done for all stacks of current interest, the contents of these stacks are moved to the top of the d^2 stacks (fourth step, marked with double arrows "$\rightarrow\rightarrow$"). After rearranging these items as needed, they can then be returned (fifth step, marked with double arrows "$\rightarrow\rightarrow$"). The "simulated" fat-column shuffle mirrors the step of permuting the first column of Fig. 3(a) to the first column of Fig. 3(d).

2. Sort the d^2 elements on top of the stacks arbitrarily, which takes $O(d^2)$ steps. This requires using the buffer stack to hold at most one item temporarily. This happens in the fifth (bottom middle) figure of Fig. 5.
3. Revert the first step above to return the sorted d^2 items to the d stacks of current interest. This corresponds going from the fifth figure to the last figure in Fig. 5.

Following the same procedure, a fat row permutation can also be carried out in $O(d^2)$ steps. To apply Proposition 2, we partition all $nd = d^3$ items into d types where items of type t, $1 \leq t \leq d$, have destinations in stack $(t-1)d+1$ to stack td. By Proposition 2, using d fat column permutations and d fat row permutations, all items of type $t, 1 \leq t \leq d$ can be moved to fat column t. Then, applying a fat column permutation to a fat column t can sort items in the fat column arbitrarily. This solves the `LSR` problem (and therefore, a `USR` problem).

Tallying the number of steps, we have done $3d$ fat column/row permutations, each of which takes $O(d^2)$ stack pop-and-pushes. The total is then $O(d^3) = O(nd)$ (with more careful counting, we can conclude that the number of stack operations is bounded by $27nd$). □

It is straightforward to see that Lemma 6 readily generalizes to $d < \sqrt{n}$. If n is a square, then the corollary directly applies. For n that is not a square, e.g., $n = m^2 + p$ where m^2 is the largest square less than n, we can partition the n stacks into two groups of m^2 stacks each with $m^2 - p$ of the stacks overlap between the two groups (We can assume that n is sufficiently large so that $m^2 - p > p$; otherwise n can be treated as a constant). Focusing on the first group of m^2 stacks, we can then apply Lemma 6 (note that m satisfies $\sqrt{n} > m > \lceil\sqrt{n}\rceil - 1 \geq d$) to "concentrate" items that should go to the rest p stacks in the $m^2 - p$ stacks shared between the two groups. Then, Lemma 6 can be applied again to the second group of m^2 stacks in a similar fashion, followed by one last application to the first group of m^2 stacks, which solves the entire problem. We have proved

Corollary 2 (Linear Step Algorithm for `LSR`, $d \leq \sqrt{n}$). *`LSR` with $d \leq \sqrt{n}$ can be solved using $O(nd)$ steps.*

Another consequence of Proposition 2 is that, if we allow $b = \lceil\sqrt{n}\rceil$ empty buffer stacks (instead of a single buffer stack) of depth d each, `USR` with arbitrary n and d can be solved using $O(nd)$ steps. This is true because a constrained (items are distinguishable by types but do not have individual label) fat column permutation can be readily executed in $2\sqrt{n}d$ steps using $\lceil\sqrt{n}\rceil$ buffer stacks.

Corollary 3 (Linear Step Algorithm for `USR` with Extra Buffers). *Given $b = \lceil\sqrt{n}\rceil$ buffer stacks, `USR` with arbitrary but sufficiently large n and d can be solved using $O(nd)$ steps.*

If n is a perfect square, then the number of required steps is bounded by $6nd$. It is not clear that having $\lceil\sqrt{n}\rceil$ buffers help with solving LSR in $O(nd)$ time for arbitrary n and d; we leave this as an open question.

4.2 Linear Step Algorithm for LSR with $d = n^{\frac{m}{2}}$ and Constant m

We continue to look at the case where $2\frac{\log d}{\log n} > 1$, starting with $n = d = k^2$ for some integer k. The algorithm for doing so will invoke Lemma 6 repeatedly, which uses the top k rows of the stacks.

Lemma 7 (Linear Step Algorithm for LSR, $d = n$). *For $n = d = k^2$, LSR can be solved in $O(nd)$ steps.*

Proof. Similar to how Lemma 6 is proven, we will simulate column and row permutations on a fat Rubik table mapped to the stack rearrangement instance. To do the mapping, we simply identify stacks $(i-1)k+1, \ldots ik$ with the i^{th} fat column of the fat Rubik table. The $j, j+k, j+2k, \ldots, j+(d-1)k$ stacks are identified with the j^{th} fat row. It is clear that, if we can simulate fat column/row permutations using $O(k^3)$ steps, then the statement of the lemma holds.

To simulate a fat column/row permutation, we note that the content of any k stacks can be flipped with the contents of the top k rows of the k^2 stacks, using the buffer stack. This takes $O(k^3)$ stack operations and is illustrated in Fig. 6(a)→(b), which is similar to the procedure illustrated in Fig. 5 (if we "compress" k consecutive items in a stack into a single item). Once the contents of the selected k stacks (corresponding to a fat column/row) occupy the top k rows of the k^2 stacks, Lemma 6 may be applied to rearrange the items in them arbitrarily, which takes $O(k^3)$ time as well. A reversal of the first step then completes a simulated fat column/row permutation. The total number of operations used is $O(k^3)$. □

It is clear that Lemma 7 continues to apply when $\sqrt{n} < d < n$, following the same argument used for establishing Corollary 3. That is,

Proposition 4 (Linear Step Algorithm for LSR, $d \leq n$). *LSR with $d \leq n$ can be solved using $O(nd)$ steps.*

The condition $d = n$ in Lemma 7 may be viewed as $\frac{\log d}{\log n} = 1$ or $d = n^{\frac{m}{2}}$ with $m = 2$. Taking a closer look at the proof for Lemma 7, it is straightforward to see that the same argument directly extends to show that the LSR case of $d = k^3$ and $n = k^2$ ($\frac{\log d}{\log n} = \frac{3}{2}$) can be solved using $O(nd)$ steps for any positive integer k. In proving Lemma 7, the top k rows of the stacks are used as a *swap space* for applying Lemma 6, simulating a fat column/row permutation. In a similar fashion, for $d = k^3$ and $n = k^2$, the top k^2 rows can be used as the swap space,

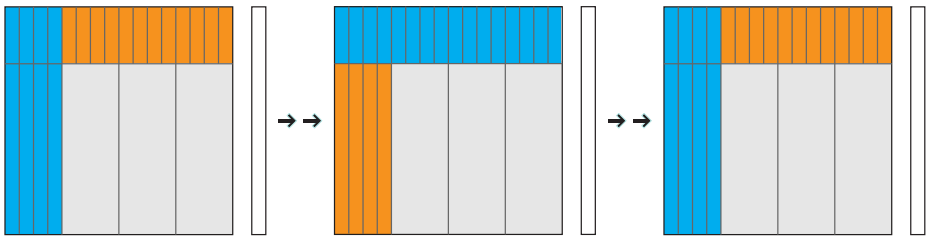

Fig. 6. Illustration of a simulated fat column permutation over n stacks of depth d, with $n = d = k^2$. (a) The cyan colored stacks map to a fat column of a Fat Rubik table. (b) Moving from the configuration given in (a) can be done in $O(k^3)$ stack pop-and-pushes. Lemma 6 can be applied to the top k rows. (c) After rearrangement, the stacks contents are restored, completing the fat column shuffle.

which allows us to work with a total of $k^2 \cdot k^2 = k^4$ items. Once the swap space is properly set up, the k^4 items can be rearranged arbitrarily by Lemma 7 using $O(k^4)$ pop-and-pushes. So LSR with $d = n^{\frac{m}{2}}$ for $m = 3$ can be solved in $O(nd)$ steps. Corollary 3 then generalizes to apply to all cases where $\frac{\log d}{\log n} \leq \frac{3}{2}$.

Recursively, Lemma 7 may be generalized to arbitrary $m \geq 2$. For $m = 3$, the procedure will call the $m = 2$ case $3k$ times. If the $n = d$ case requires $cnd = ck^4$ steps for some constant c, then the $m = 3$ case will need $3ck^5$ steps. Recursively, for general m, the recursive procedure will require about $3^m cnd$ steps for $d = n^{\frac{m}{2}}$. We have proved

Theorem 1 (Algorithm for LSR with $d = n^{\frac{m}{2}}$ and $m \geq 2$). *LSR with $d = n^{\frac{m}{2}}$ for $m \geq 2$ can be solved using $O(3^m nd)$ steps.*

For any fixed $m \geq 2$, it is clear that LSR can be solved in $O(nd)$ steps for $n^{\frac{m-1}{2}} < d < n^{\frac{m}{2}}$, possibly with a larger constant than the $d = n^{\frac{m}{2}}$ case. For fixed m, 3^m is also a constant. Summarizing the results on the upper bounds obtained so far, we have

Theorem 2 (Linear Step Algorithm for LSR with $d \leq \lceil cn \rceil$). *For arbitrary fixed real number $c > 0$, LSR with $d \leq \lceil cn \rceil$ can be solved using $O(nd)$ steps.*

For USR with $d = n = k^2$, with additional care in carrying out the recursive procedure, we only need to make $2k$ calls to Lemma 6 instead of $3k$ as required in proving Lemma 7. This gives us that USR with $d = n^{\frac{m}{2}}$ for $m \geq 2$ can be solved using $O(2^m nd)$ steps instead of the $O(3^m nd)$ stated in Theorem 1. We omit the very involved procedure, which boils down to doing a mixed column and row permutation. The procedure does not apply to LSR.

4.3 Constant n or d

Lastly, we briefly discuss what happens when n or d is a constant. An $O(nd \log n)$ algorithm for USR is provided in [15] for arbitrary n and d, using divide and conquer over the number of stacks n. This implies that for constant n, $O(d)$ steps is sufficient, matching the $\Omega(nd)$ lower bound. For constant d, each stack can be sorted in $O(1)$ steps by first moving all type k items to the top of the stacks they are at (for a stack i that contains type k item, this can be done by first moving the top item from some d stacks to the buffer, moving items in stack i to the empty d top spots, and then moving them back to stack i so that type k items stay on the top). Then type k items can be all moved to the buffer stack and followed by emptying stack k, then to stack k. This yields an $O(n)$-step algorithm, also matching the lower bound.

5 Conclusion and Discussion

In this study, we have analyzed a formulation of the stack rearrangement problem where objects stored in stacks must be shuffled. A stack can only be accessed from the top (i.e., it is a LIFO queue). As the main result, we show that the labeled and unlabeled versions of the problem with n filled stacks of capacity d can both be solved using $O(nd)$ (i.e., linear number of) steps for an average case input, where $d \leq \lceil cn \rceil$ for some constant c. This closely matches the lower bound $O(nd)$ for USR and LSR (when $d \leq \lceil cn \rceil$, $\frac{\log d}{\log n}$ is a constant).

We conclude the work by raising several open questions.

Bound Gap. Whereas we know that it is not possible to reach $O(nd)$ for LSR for arbitrary n and d, we do not know whether the same is true for USR. In our algorithmic solution, though we achieve $O(nd)$ for arbitrarily large but fixed $\frac{d}{n}$, we have not fully closed the gap. In the approach that we have used, the issue is caused by the $3k$ recursive calls. The 3 there is where the 3^m factor (in the $O(3^m nd)$ complexity stated in Theorem 1) comes from. For USR, we were able to future drop the required number of moves to $O(2^m nd)$. Reducing the number of recursive calls may get us closer to closing the small remaining gap between the lower and upper bounds.

Hardness. The question of whether USR and LSR are NP-hard to solve optimally remains open. In this regard, it may be interesting to study the case of constant d. Whereas the case of $d = 1$ can be readily solved, larger d appears to be challenging.

Utility of Multiple Buffer Stacks In the current study, we have mainly examined the case of using a single buffer stack. We also show that using $\sqrt{n}$ empty buffer stacks allow the resolution of USR in $O(nd)$ steps. A natural question to ask is for what values of $b \in [1, \sqrt{n})$, b empty buffer stacks would enable solving USR in $O(nd)$ steps. As have been discussed, it is not clear that $\sqrt{n}$ buffer stacks

are sufficient for solving LSR in $O(nd)$ steps for arbitrary n and d, which also warrants further examination.

Other Queuing Models As generalizations to the current problem, it could be interesting to study a two-dimensional stack setting, e.g., items may be accessed both from the top or from the left side. Does such a setting, which provides similar storage capacity as stacks, allows more access flexibility? One may also replace a stack with a queue that may be accessed from both ends. Many additional settings similar to these two can be examined.

Acknowledgement

M. Szegedy is with the Alibaba Quantum Laboratory, Alibaba Group, Bellevue, WA 98004, USA. The research was done while he was working at Rutgers University. J. Yu is with the Department of Computer Science, Rutgers, the State University of New Jersey, Piscataway, NJ 08854, USA. E-Mails: {szegedy, jingjin.yu} @ cs.rutgers.edu. The work is supported in part by NSF awards IIS-1617744, IIS-1734419, IIS-1845888 and CCF-1934924.

References

1. Ajtai, M., Komlós, J., Szemerédi, E.: An 0 (n log n) sorting network. In: Proceedings of the fifteenth annual ACM symposium on Theory of computing, pp. 1–9 (1983)
2. Ben-Shahar, O., Rivlin, E.: Practical pushing planning for rearrangement tasks. IEEE Transactions on Robotics and Automation **14**(4), 549–565 (1998)
3. Borgman, B., van Asperen, E., Dekker, R.: Online rules for container stacking. OR spectrum **32**(3), 687–716 (2010)
4. Brousseau, B.A.: Tower of hanoi with more pegs. J. Recreational Mathematics **8** (1980)
5. Christofides, N., Colloff, I.: The rearrangement of items in a warehouse. Operations Research **21**(2), 577–589 (1973)
6. Cole, R., Ost, K., Schirra, S.: Edge-coloring bipartite multigraphs in o (e logd) time. Combinatorica **21**(1), 5–12 (2001)
7. Dayama, N.R., Krishnamoorthy, M., Ernst, A., Narayanan, V., Rangaraj, N.: Approaches for solving the container stacking problem with route distance minimization and stack rearrangement considerations. Computers & Operations Research **52**, 68–83 (2014)
8. Demaine, E.D., Demaine, M.L., O'Rourke, J.: Pushpush and push-1 are np-hard in 2d. arXiv preprint cs/0007021 (2000)
9. Demaine, E.D., Hoffmann, M.: Pushing blocks is np-complete for noncrossing solution paths (2001)
10. Erdmann, M., Lozano-Perez, T.: On multiple moving objects. Algorithmica **2**(1-4), 477 (1987)
11. Garrett, C.R., Lozano-Pérez, T., Kaelbling, L.P.: Ffrob: An efficient heuristic for task and motion planning. In: Algorithmic Foundations of Robotics XI, pp. 179–195. Springer (2015)

12. Goel, A., Kapralov, M., Khanna, S.: Perfect matchings in o(n\logn) time in regular bipartite graphs. SIAM Journal on Computing **42**(3), 1392–1404 (2013)
13. Grigorchuk, R., Šunik, Z.: Asymptotic aspects of schreier graphs and hanoi towers groups. Comptes Rendus Mathematique **342**(8), 545–550 (2006)
14. Hall, P.: On representatives of subsets. In: Classic Papers in Combinatorics, pp. 58–62. Springer (2009)
15. Han, S.D., Stiffler, N.M., Bekris, K.E., Yu, J.: Efficient, high-quality stack rearrangement. IEEE Robotics and Automation Letters **3**(3), 1608–1615 (2018). Note: presented at ICRA 2018
16. Han, S.D., Stiffler, N.M., Krontiris, A., Bekris, K.E., Yu, J.: Complexity results and fast methods for optimal tabletop rearrangement with overhand grasps. The International Journal of Robotics Research **37**(13-14), 1775–1795 (2018)
17. Havur, G., Ozbilgin, G., Erdem, E., Patoglu, V.: Geometric rearrangement of multiple movable objects on cluttered surfaces: A hybrid reasoning approach. In: 2014 IEEE International Conference on Robotics and Automation (ICRA), pp. 445–452. IEEE (2014)
18. Huang, E., Jia, Z., Mason, M.T.: Large-scale multi-object rearrangement. In: 2019 International Conference on Robotics and Automation (ICRA), pp. 211–218. IEEE (2019)
19. Krontiris, A., Bekris, K.E.: Dealing with difficult instances of object rearrangement. In: Robotics: Science and Systems (2015)
20. Krontiris, A., Bekris, K.E.: Efficiently solving general rearrangement tasks: A fast extension primitive for an incremental sampling-based planner. In: 2016 IEEE International Conference on Robotics and Automation (ICRA), pp. 3924–3931. IEEE (2016)
21. Shome, R., Solovey, K., Yu, J., Bekris, K., Halperin, D.: Fast, high-quality dual-arm rearrangement in synchronous, monotone tabletop setups. arXiv preprint arXiv:1810.12202 (2018)
22. Solovey, K., Halperin, D.: On the hardness of unlabeled multi-robot motion planning. The International Journal of Robotics Research **35**(14), 1750–1759 (2016)
23. Spiralris, D.K.G.M.P.: Coordinating pebble motion on graphs, the diameter of permutation groups, and applications (1984)
24. Stilman, M., Kuffner, J.: Planning among movable obstacles with artificial constraints. The International Journal of Robotics Research **27**(11-12), 1295–1307 (2008)
25. Szegedy, M.: In how many steps the k peg version of the towers of hanoi game can be solved? In: Annual Symposium on Theoretical Aspects of Computer Science, pp. 356–361. Springer (1999)
26. Van Den Berg, J., Stilman, M., Kuffner, J., Lin, M., Manocha, D.: Path planning among movable obstacles: a probabilistically complete approach. In: Algorithmic Foundation of Robotics VIII, pp. 599–614. Springer (2009)
27. West, J.: Sorting twice through a stack. Theoretical Computer Science **117**(1-2), 303–313 (1993)
28. Wilfong, G.: Motion planning in the presence of movable obstacles. Annals of Mathematics and Artificial Intelligence **3**(1), 131–150 (1991)
29. Yu, J.: Constant factor time optimal multi-robot routing on high-dimensional grid. In: Robotics: Science and Systems (RSS) (2018)